

# JS Level 0



# Введение



# Введение

Сегодняшняя наша лекция снова будет посвящена продолжению работы с CSS.

Мы обсудим ключевые моменты работы CSS:

- каскадность
- селекторы
- блочную модель
- позиционирование

Все эти знания (и ряд других) нужны нам для того, чтобы полноценно верстать макеты страниц (и, в дальнейшем) работать со страницей из JS.



# Нагрузка

**Важно:** потихоньку нужно привыкать к увеличивающемуся количеству материала, которое вам нужно знать и запоминать. Мы немного усложним вам задачу, разместив в одной лекции достаточно много теории (нужно понимать, что если вы хотите быть разработчиком, то каждый день вы будете изучать что-то новое, поэтому нужно заранее "приучать" мозг к ежедневной обработке новой информации).



# Каскад



# CSS

Когда мы начинали знакомиться с CSS, то давали следующее определение: CSS (Cascading Style Sheets) – каскадные таблицы стилей. Но что значит каскадные?

```
1  <!DOCTYPE html>
2  <html lang="ru">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Алиф</title>
7      <link rel="stylesheet" href="normalize.min.css">
8      <link rel="stylesheet" href="fonts/roboto.css">
9      <link rel="stylesheet" href="style.css">
10 </head>
11 <body>
12     <header>
13         <nav>
14             <a href="https://alif.tj">
15                 
16             </a>
17             <ul>
18                 <li><a href="https://alifshop.tj">alif shop</a></li>
19                 <li><a href="https://intiqol.tj">Переводы</a></li>
20                 <li><a href="https://deposit.alif.tj">Депозиты</a></li>
21                 <li><a href="https://visa.alif.tj">Visa</a></li>
22             </ul>
23         </nav>
24     </header>
25 </body>
26 </html>
```



# CSS

Первое, с чего мы начнём (и уже это обсуждали) – стили могут конфликтовать друг с другом. Например:

```
a {  
  color: ■ red;  
}  
a {  
  color: ■ blue;  
}
```

В этом простейшем случае выигрывает тот стиль, который идёт ниже по документу (т.е. последний).

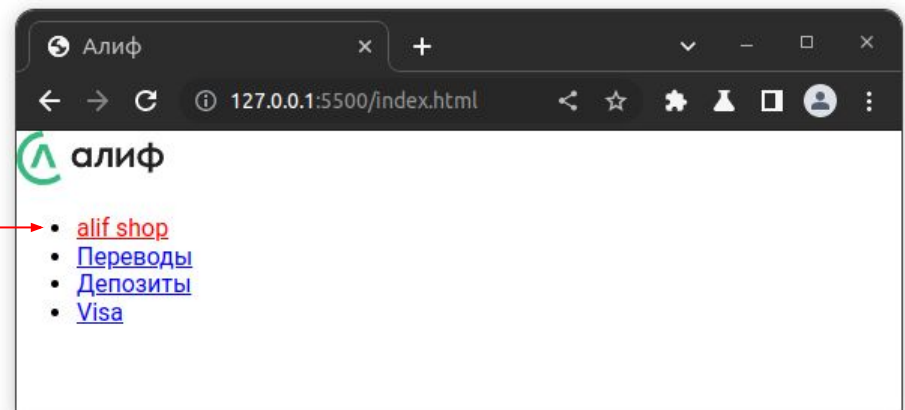


# CSS

Но это, конечно же, не всё. Оказывается, мы можем с вами писать более сложные селекторы, которые более точно выбирают элементы и исходя из этого правило "выигрывает последний" не будет работать. Давайте посмотрим:

```
[href="https://alifshop.tj"] {  
  color: ■ red;  
}  
a {  
  color: ■ blue;  
}
```

выигрывает для конкретной ссылки





# Селекторы

Селекторы – достаточно сложный язык выражений, которому посвящена [целая спецификация](#).

Мы с вами обсудим ключевые правила составления селекторов, которыми вы будете пользоваться чаще всего.



# \* универсальный селектор

Первый селектор, который у нас есть – это универсальный селектор (\*), позволяет выбирать все элементы на странице:

```
* {  
  color: ■ red;  
}
```



# Селектор по типу (тегу)

Второй вид селекторов – селекторы по типу (или тегу), позволяют выбирать только элементы, относящиеся к определённому типу (мы их уже использовали на прошлом занятии):

```
a {  
  color: red;  
}
```



# Селекторы по атрибутам

А дальше – интереснее, мы можем выбирать элементы по атрибутам: т.е. выбирать только те элементы, у которых есть определённые атрибуты:

любой элемент с атрибутом `href`, равен <https://alifshop.tj>



```
[href="https://alifshop.tj"] {  
  color: red;  
}
```

Т.е. мы в квадратных скобках пишем `[название_атрибута=значение_атрибута]`.

Рекомендуется включать значение атрибута в кавычки (иначе не будет работать, если значение содержит, например, пробел).



# Селекторы атрибутов

И это, конечно же не всё, есть и более продвинутые варианты (мы их приводим только для ознакомления, использовать мы их будем только в особых случаях):

`[attr]` – у элемента есть атрибут `attr` (значение нам не важно)

`[attr="val"]` – у элемента есть атрибут `attr` со значением `val` (см. предыдущий слайд)

`[attr~="val1 val2"]` - у элемента есть атрибут `attr` с одним из значений `val1` или `val2` (значения должны быть разделены пробелами)

`[attr|=val]` – у элемента есть атрибут `attr` со значением `val` (точное совпадение) или начинающимся на `val-`

`[attr^=val]` – у элемента есть атрибут `attr` со значением, начинающимся с `val`

`[attr$=val]` – у элемента есть атрибут `attr` со значением, заканчивающимся на `val`

`[attr*=val]` – у элемента есть атрибут `attr` со значением, содержащим подстроку `val`



# "Вес" селекторов

Q: зачем мы обсуждали селекторы по атрибуту?

A: затем чтобы ввести понятие "веса" селектора

Правила:

1. Селектор типа "стоит" 1
2. Селектор атрибута стоит "10"

Но это ещё не всё. Есть два специальных атрибута (и их мы будем использовать чаще всего).



# class

Атрибут `class` – позволяет через пробел задать список значений, с помощью которых затем можно стилизовать элемент (или группу элементов). С помощью классов мы можем объединять разнородные элементы в группу, чтобы стилизовать их одинаково:

у `a` один класс

```
<a class="button" href="https://alif.tj">Контакты</a>  
<button class="button primary">Отправить</button>
```

у `button` два  
класса

Здесь важно то, что у кнопки и у ссылки есть один общий класс, чтобы визуально их представлять как кнопку.



# class

Пример из популярного фреймворка Bootstrap:

Link

Button

Input

Submit

Reset

HTML

```
<a class="btn btn-primary" href="#" role="button">Link</a>
<button class="btn btn-primary" type="submit">Button</button>
<input class="btn btn-primary" type="button" value="Input">
<input class="btn btn-primary" type="submit" value="Submit">
<input class="btn btn-primary" type="reset" value="Reset">
```

Как мы видим, элементы совершенно разные, но благодаря оформлению на основе классов визуально они выглядят одинаково (при этом сохраняется их семантика).





# class

Как мы видим, для селектора класса придумали свой специальный синтаксис – через точку (хотя и как для атрибута тоже будет работать):

```
.button {  
  background-color: ■ #ff0000;  
  color: □ #ffffff;  
}
```

Именно селектор по классу мы будем использовать больше всего (это достаточно частая практика). "Вес" селектора по классу такой же, как и у селектора по атрибуту.

Кстати, в Emmet класс тоже задаётся через точку, например:

`button.button.primary` + Tab



# id

И последний специализированный селектор по атрибуту – `id`:

```
<a id="logo" href="https://alif.tj">  
    
</a>
```

```
#logo {  
  
}
```

Логически предполагается, что на странице может быть только один элемент с `id="logo"` (по факту – может быть не так, и браузер простит нам это), поэтому это позволяет уникальным образом выбрать элемент на странице.

Кстати, `id` в Emmet так и задаётся через `#`:

```
a#logo[href="https://alif.tj"]
```



# id

Вес селектора по `id = 100`, таким образом, он имеет больший приоритет, чем те селекторы, что мы проходили.



# id и class

`id` и `class` – это специальные атрибуты, которые можно назначить любому элементу:

*DOM* defines the user agent requirements for the `class`, `id`, and `slot` attributes for any element in any namespace. [\[DOM\]](#)

The `class`, `id`, and `slot` attributes may be specified on all [HTML elements](#).



# Промежуточные итоги

Таким образом, даже если правило расположено выше, но имеет более высокий вес, то выиграет оно, а не то, которое ниже.

```
#logo {  
  /* выиграет у нижестоящего для элемента с id=logo */  
}
```

```
a {  
  /* */  
}
```



# Промежуточные итоги

Важные правила:

1. Когда мы пишем свои правила, мы стараемся идти от более общего к более частному (т.е. надо поменять правила в примере выше)
2. Но иногда нам приходится переопределять чужие правила, тогда мы свои пишем ниже и учитываем, что вес наших правил должен быть равен (тогда они "выиграют" за счёт более низкого расположения) или выше (тогда они "выиграют" за счёт веса) – именно поэтому мы сначала подключали `normalize.css` и шрифты, а наш файл – в последнюю очередь



# style

А теперь давайте ещё раз вспомним то, как мы можем задавать стили (свои, которые переопределяют стили браузера):

1. Элемент **style**
2. Внешний файл
3. Атрибут **style** у элемента



# style

Так вот нас интересует последний вариант, он означает, что мы можем назначить элементу стиль прямо в теге:

```
<li>
  <a id="shop" href="https://alifshop.tj" style="color: ■ green;">
    alif shop
  </a>
</li>
```

Результат:

```
#shop {
  color: ■ red; ← это в style.css
}
```

- [alif shop](#)
- [Переводы](#)
- [Депозиты](#)
- [Visa](#)

У него нет селектора и вес подобного правила равен 1000.





# Веса

Если составить табличку, то получится что (как мы видим, сплошная математика):

правило	вес
в атрибуте <code>style</code>	1000
по селектору <code>id</code>	100
по селектору класса или атрибута	10
по селектору тега	1

"Побеждает" тот стиль (если правильно говорить, задаваемое стилем свойство), у которого выше "вес" (или, если правильно выразаться, специфичность). Если же специфичность одинаковая, то выигрывает то правило, которое объявлено позже. За одним исключением, давайте посмотрим, каким.



# Веса

Тут прямо важно уточнить:

```
#logo {
```

```
  color: red;
```

```
}
```

```
a {
```

```
  color: blue;
```

```
  background-color: transparent;
```

```
}
```

конфликт только здесь



Если у нас элемент `<a id="logo" ...>...</a>`, то в этом случае конфликтуют у нас только стили, относящиеся к цвету текста (с фоновым цветом никаких конфликтов нет). Про наследование мы поговорим чуть позже.



# !important

Последняя конструкция, которую мы обсудим и которую вам нужно всячески избегать – это **!important** (его очень любят начинающие, т.к. вместо того, чтобы разбираться, почему что-то не так, они просто лепят везде **!important**, если бы можно было **!important** писать несколько раз подряд и это бы добавляло "веса", они бы так и делали).

Что делает **!important**? **!important** говорит, что отмеченное ею стиль важнее всех остальных:

```
<li>
  <a id="shop" href="https://alifshop.tj" style="color: ■green;">
    alif shop
  </a>
</li>
```

```
#shop {
  color: ■red !important;
}
```

- [alif shop](#)
- [Переводы](#)
- [Депозиты](#)
- [Visa](#)



# !important vs !important

А что если встретятся два **!important**? Тогда выиграет тот, у кого специфичнее правило. А если и правила одинаковы по специфичности, то тот, который последний.

Итого получаем алгоритм:

1. **!important** – если у одного правила есть **!important**, а у другого – нет, то выигрывает то, у которого есть, и дальше не смотрим
2. Специфичность (вес) – если у обоих правил есть **!important** или у обоих нет, то смотрим вес, у кого больше – тот и выигрывает, дальше не смотрим
3. Порядок – если вес одинаковый (с учётом **!important** – у обоих есть или у обоих нет), то смотрим, какое правило идёт последним



# На практике

Чтобы не запоминать с предыдущей страницы (а тем более не вычислять специфичность), чаще всего стараются обойтись селекторами по `id` и классами, а также атрибутом `style` (но с ним работают уже через JavaScript).

**Но:** вам достаточно часто придётся иметь дело с "творчеством" других разработчиков (и своим тоже), и достаточно часто придётся разбираться, почему одно правило "выигрывает" у другого. Поэтому лучше правила запоминать.



# Комбинации

На самом деле, мы почти закончили с каскадом, остался лишь последний момент: селекторы можно комбинировать (за счёт этого в том числе повышается их специфичность).



# Комбинации

Допустим, у нас есть селекторы **A** и **B**. Мы можем их комбинировать следующим образом:

- **AB** – элементы, одновременно подпадающие под действие и селектора **A**, и селектора **B**, пример:

**a.button { ... }** – элемент **a**, у которого есть класс **button**, вес будет  $1 + 10 = 11$

- **A, B** – элементы, подпадающие под действие селектора **A** и/или **B**, пример:

**a, button { ... }** – либо элемент **a**, либо элемент **button**, вес каждого селектора будет **1** (не складывается, т.к. это разные селекторы)

- **A B** – один из родителей – **A**, дочерний – **B**, пример:

**#partners li { ... }** – элементы **li** внутри элемента с **id="partners"** (на любой глубине вложенности), вес будет  $100 + 1 = 101$

- **A > B** – непосредственный родитель **A**, дочерний - **B**, пример:

**#partners > li { ... }** – элементы **li** внутри элемента с **id="partners"** (**li** должен быть непосредственным ребёнком, например: `<ul id="partners"><li>...</li></ul>`), вес будет  $100 + 1 = 101$



# Комбинации

При этом стоит отметить, что **A** и **B** сами могут быть сколь угодно сложными, например, пример с <https://ya.ru> (как видите, они обходятся классами):

```
.input_ahead-visible_yes .input__ahead, .input_theme_websearch .input__control {  
    display: block;  
}
```

Как вы видите, тут форма A, B, где:

- A – `.input_ahead-visible_yes .input__ahead`
- B – `.input_theme_websearch .input__control`

Можете посмотреть с помощью [специального калькулятора](#) веса.





# Каскад

Вернёмся к понятию каскад (из каскадные таблицы стилей): каскад – это просто алгоритм (последовательность шагов), которая сортирует конфликтующие правила по приоритетам и выбирает только то, у которого получился самый большой приоритет.



# Промежуточные итоги

Мы с вами поговорили о том, какие самые распространённые селекторы существуют, как вычисляется то, какое правило будет применяться при конфликте правил и про комбинации селекторов.

Это достаточно простые правила, которые вам нужно знать, как веб-разработчикам.



# А как же наследование?

У наследуемых свойств нулевая специфичность, что тоже нужно учитывать при задании стилей.



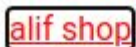
# Псевдоклассы



# Псевдоклассы

Помимо рассмотренных нами селекторов, есть также специальные селекторы псевдоклассов, которые позволяют искать элементы с определённым состоянием или по другим свойствам (т.е. псевдокласс – это буквально класс, который задан не вами, а определяется браузером в ответ на что-то, например, состояние или позицию элемента среди детей родителя – первый, последний, нечётный).

Например, у гиперссылки есть целых 5 состояний:

- **:link** – непосещённая ссылка (т.е. по ней ещё не переходили)
- **:visited** – ссылка была посещена (т.е. по ней уже переходили)
- **:hover** – на ссылку навели указатель мыши
- **:active** – на ссылку нажали
- **:focus** – на ссылку установили **focus** (с помощью клавиши **Tab**):
  -  **alif shop** ← пример установки фокуса
  - [Переводы](#)
  - [Депозиты](#)
  - [Visa](#)



# Псевдоклассы

Обычно их располагают именно в таком порядке (в каком они описаны на предыдущем слайде).

Попробуйте задать следующие правила:

```
a:link {  
    color: red;  
}  
a:visited {  
    color: blue;  
}
```

И т.д., и посмотрите, как поменялось взаимодействие с элементом.



# Псевдоклассы

Список псевдоклассов вы можете найти на странице MDN:

<https://developer.mozilla.org/ru/docs/Web/CSS/Pseudo-classes>



# Псевдоэлементы





# Псевдоэлементы

Помимо псевдоклассов, у нас есть также [псевдоэлементы](#) – это возможность стилизовать определённую часть элемента.

Ключевыми псевдоэлементами будут являться:

- `::before` – до вашего элемента
- `::after` – после вашего элемента

**Важно:** это именно псевдоэлементы, а не предыдущий и следующий элемент в документе.



# Псевдоэлементы

Посмотрим на практике:

style.css

```

.selected::marker {
  color: red;
}
.selected::before {
  content: ">";
}
.selected::after {
  content: ">";
}

```

```

▼ <ul>
  ► <li>...</li>
  ▼ <li class="selected">
    ► ::marker
    ► ::before
      "второй"
    ► ::after
      </li>
  ► <li>...</li>
  </ul>

```

- первый
- >второй<
- третий

```

<body>
  <ul>
    <li>первый</li>
    <li class="selected">второй</li>
    <li>третий</li>
  </ul>
</body>

```

Свойство **content** позволяет задать содержимое псевдоэлементу.



# Поток документа и блочная модель



# Блочная модель

Для того, чтобы двигаться дальше, нам необходимо обсудить блочную модель и поток документа. Давайте разбираться, что это такое.



# display

За то, к какому типу будет принадлежать элемент, отвечает свойство [display](#).

Ключевые значения для нас:

- inline
- block
- inline-block
- flex
- grid

Это не все, см. полный список по ссылке.



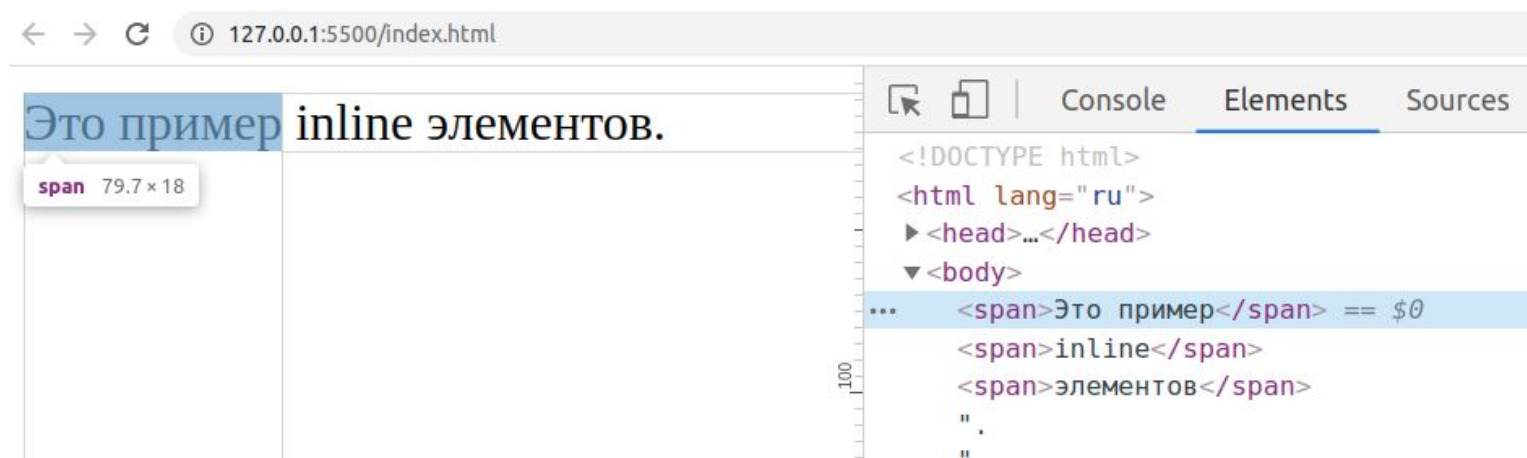
# Общая идея

Давайте посмотрим на следующий документ:

```
<body>  
| <span>Это пример</span> <span>inline</span> <span>элементов</span>.  
</body>
```

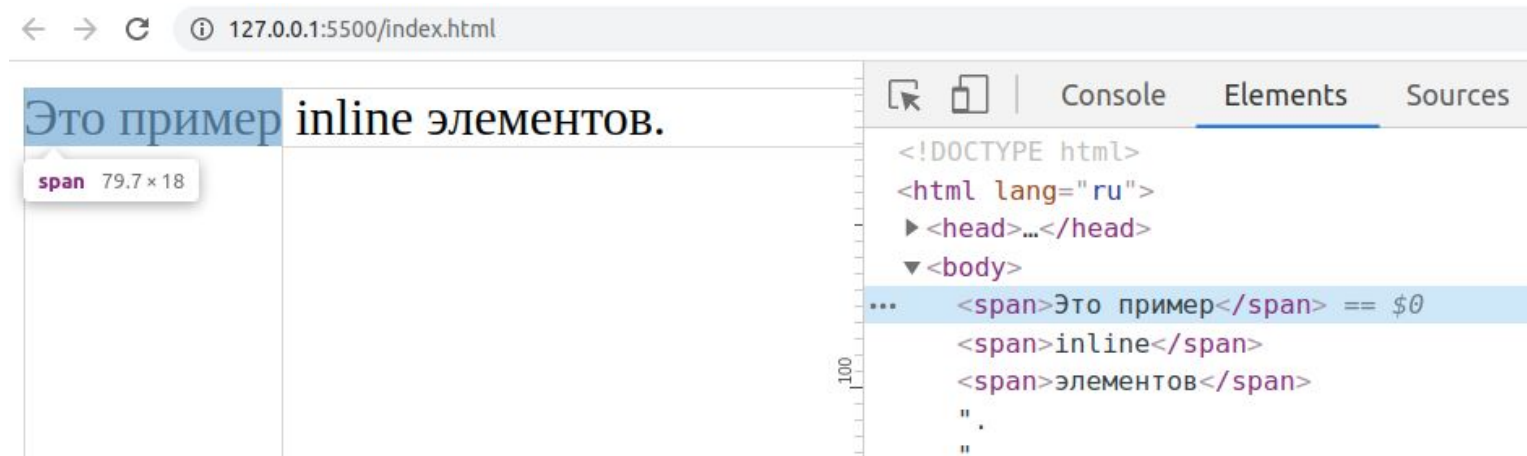
возможно, вам придётся нажать ещё **Fn** (если у вас ноутбук)

Откроем DevTools (**F12** или **Ctrl + Shift + I**) и перейдём на вкладку **Elements**. Если мы теперь начнём наводить курсором на элементы (во вкладке **Elements**), то увидим, что элементы размещены "в строку" (**inline**):



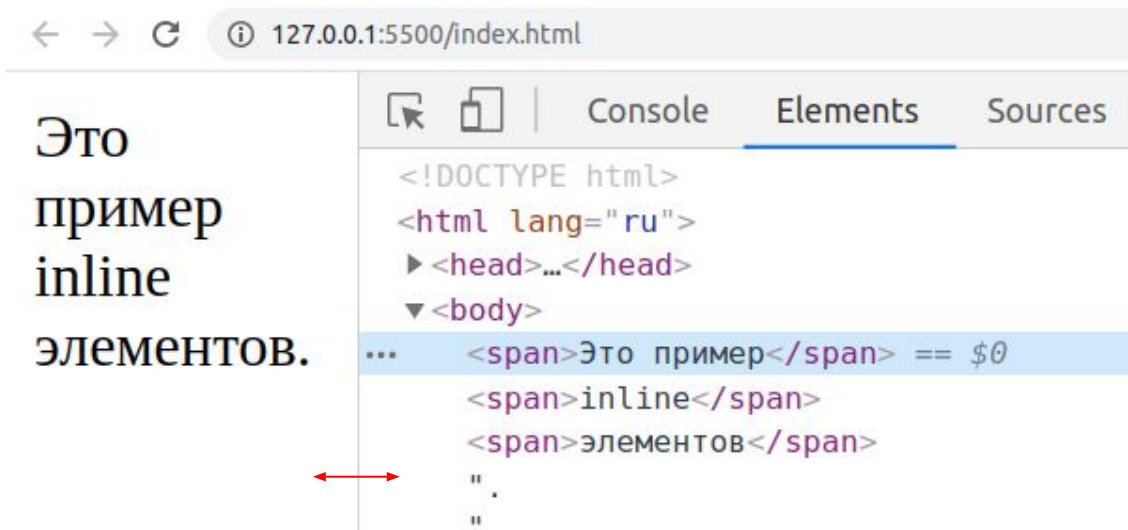
# Общая идея

Каждый элемент занимает ровно столько места, сколько требует его содержимое (тот текст, который он содержит):



# Общая идея

Если мы начнём передвигать границу (чтобы элементы не умещались на одной строке), то они начнут переходить на следующую строку:



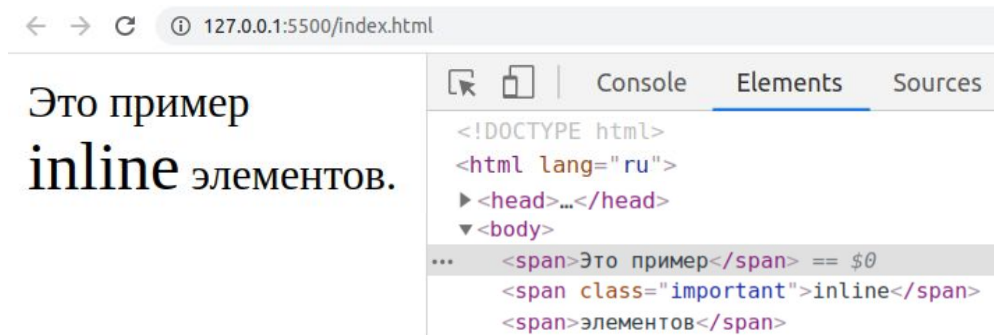
Причём, если элемент не умещается на одной строке, то его вполне может "разорвать" на две строки.





# Общая идея

Давайте изменим размер шрифта одного из элементов и посмотрим, что получится:



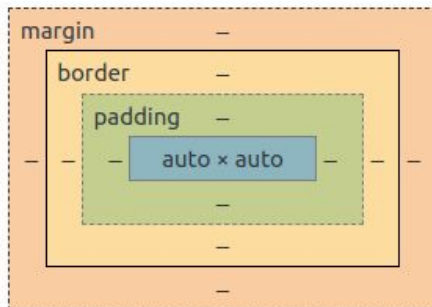
Сама "виртуальная строка" увеличится (именно вторая), чтобы уместить целиком элемент с большим размером шрифта.



# Поток документа

Таким образом: элементы располагаются в некотором потоке, который имеет направление слева-направо сверху-вниз (то, как мы привыкли читать и писать). И располагаются элементы ровно друг за другом, занимая отведённое им место.

По умолчанию, элемент `span` является `inline`-элементом:



При этом значением `height` и `width` является `auto` (т.е. автоматически рассчитываемое).

Давайте попробуем поменять его, а также разберёмся с нарисованной цветной схемой.

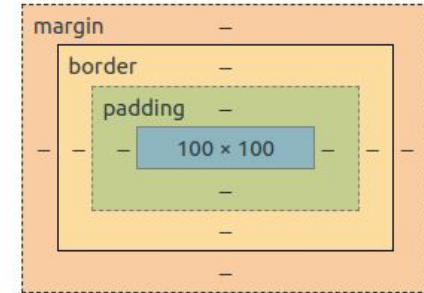
Filter	<input type="checkbox"/> Show all <input type="checkbox"/> Group
display	inline
height	auto
width	auto



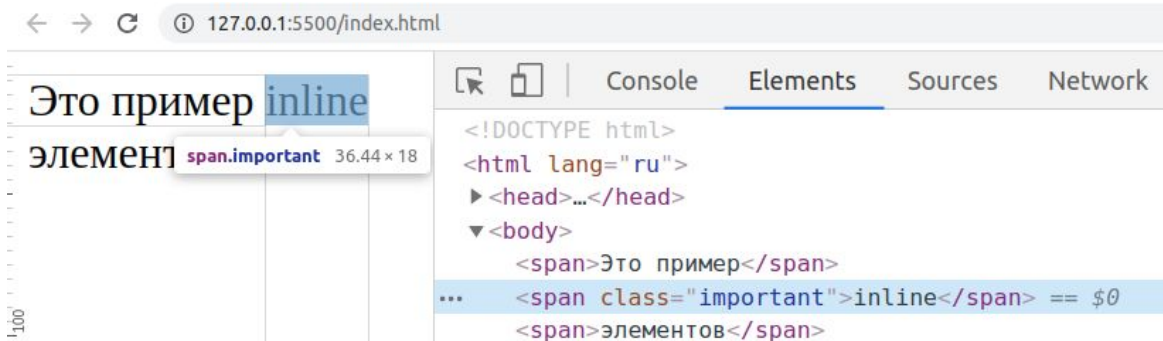
# inline

Попробуем менять размеры:

```
style.css
.important {
  height: 100px;
  width: 100px;
}
```



Filter		<input type="checkbox"/> Show all	<input type="checkbox"/> Group
display	inline		
▼ height	100px		
100px	.important		
▼ width	100px		
100px	.important		



Ничего не получилось. Почему? Потому что браузер игнорирует выставленные для **inline**-элементов высоту и ширину.



# block

Значит есть те, для кого не игнорирует? Да, действительно так. Сделаем элементы блочными:

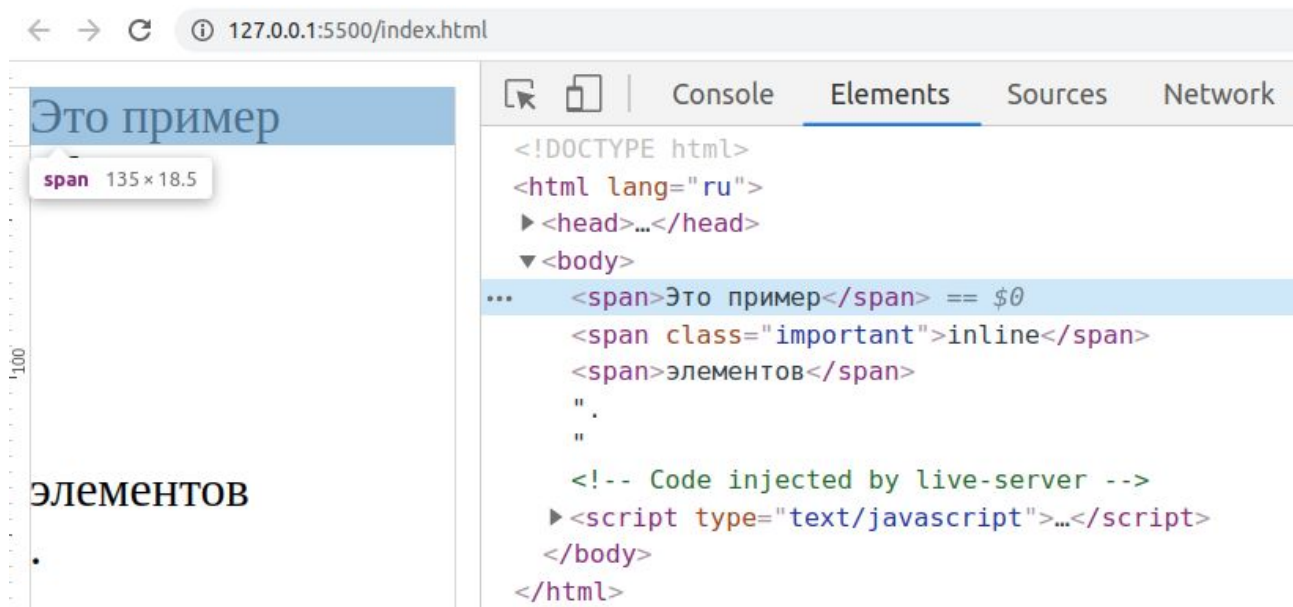
```
style.css
span {
  display: block;
}
.important {
  height: 100px;
  width: 100px;
}
```

```
<body>
  <span>Это пример</span> <span class="important">inline</span> <span>элементов</span>.
</body>
```



# block

Картинка резко поменяется: элементы начнут занимать всю доступную ширину и располагаться друг за другом уже по вертикали (т.е. не дают друг другу встать на одной строке):



# block

И самое важное: свойства `width` и `height` начнут работать:

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5500/index.html`. The browser's developer tools are open, showing the `Elements` panel. The HTML structure is as follows:

```
<!DOCTYPE html>
<html lang="ru">
  <head>...</head>
  <body>
    <span>Это пример</span>
    <span class="important">inline</span> == $0
    <span>элементов</span>
    "
    "
    <!-- Code injected by live-server -->
    <script type="text/javascript">...</script>
  </body>
</html>
```

The `span` element with `class="important"` is highlighted in blue. The browser's status bar at the bottom shows "ЭЛЕМЕНТОВ" and a single bullet point.



# inline vs block

Промежуточные итоги:

- **inline** – занимают столько места, сколько требует их содержимое, выстраиваются в строку
- **block** – занимают всю строку по ширине, а по высоте ровно столько, сколько требует содержимое (при этом можно задавать размеры)



# Блочная модель

Теперь давайте разбираться со схемой:

Styles   Computed   Layout   Event Listeners   >>

The diagram illustrates the box model with four nested layers: a blue content box (100x100), a green padding layer, a yellow border layer, and an orange margin layer. Dashed lines and tick marks indicate the boundaries and spacing between these layers.

Filter ☐ Show all ☐ Group

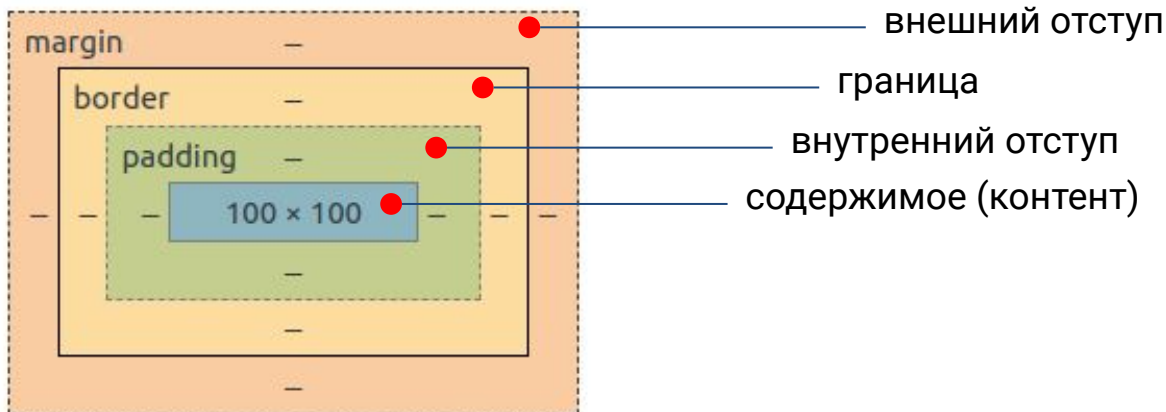
- ▶ display block
- ▶ height 100px
- ▶ width 100px





# Блочная модель

Любой элемент представлен в виде блока, состоящего из 4 областей:



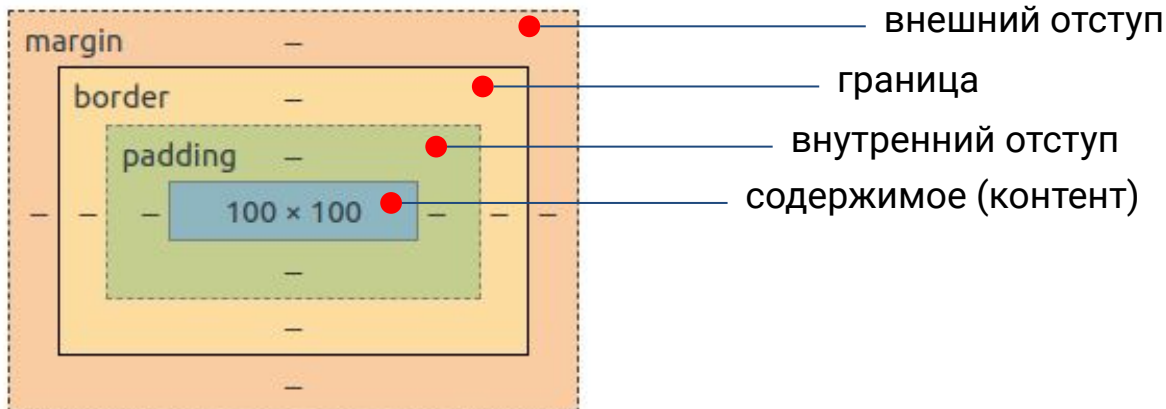
Там, где прочерки – значение не задано. Но вот для контента значение **100x100** – т.е. содержимое элемента занимает **100px** по ширине и **100px** по высоте (напоминаем, для **inline** заданные значения игнорируются и используется **auto**).



# Блочная модель

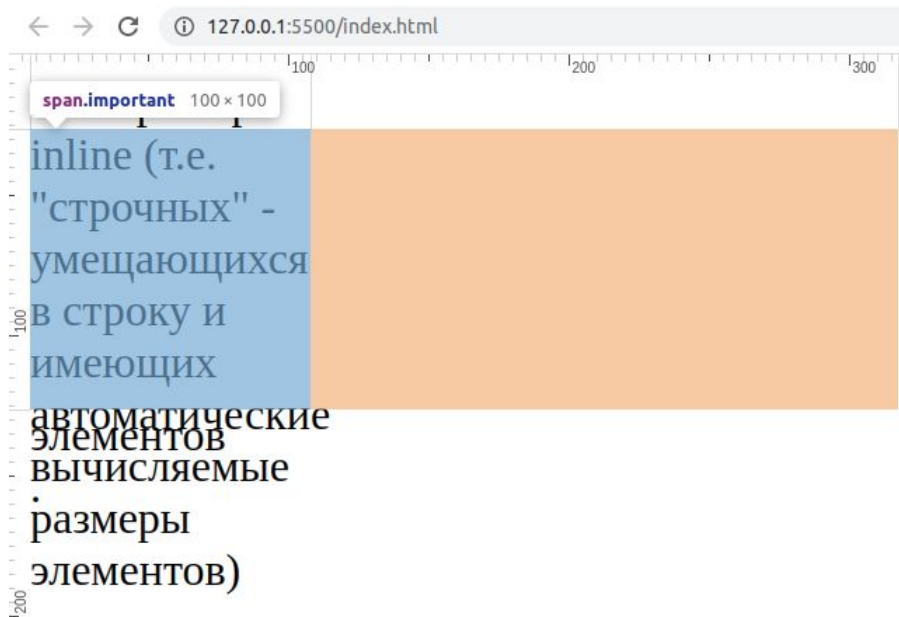
Первый вопрос, который у нас может возникнуть: зачем столько областей элементу? Ответ достаточно простой: каждая область отвечает за свою задачу:

- **margin** – какую "дистанцию" держит элемент по отношению к другим элементам
- **border** – "рамка" элемента (как у картины)
- **padding** – насколько содержимое "отталкивается" от рамки
- **content** – само содержимое



# content

Первая область, выделена синим – это **content**. Т.е. непосредственное содержимое элемента. Элемент автоматически расширяется, чтобы уместить в себе всё содержимое. Если же вы жёстко задали размер элемента (для **block**), то содержимое начинает переполнять (**overflow**) элемент:



# overflow

По умолчанию, элемент переполняется "вниз", пытаюсь разбить содержимое по словам так, чтобы они переносились на следующую строку.



# overflow

За то, как будет обрабатываться переполнение, отвечает свойство **overflow**:

## 11.1.1 Overflow: the 'overflow' property

### **'overflow'**

<i>Value:</i>	visible   hidden   scroll   auto   <u>inherit</u>
<i>Initial:</i>	visible
<i>Applies to:</i>	block containers
<i>Inherited:</i>	no
<i>Percentages:</i>	N/A
<i>Media:</i>	<u>visual</u>
<i>Computed value:</i>	as specified

Поэкспериментируйте с разными значениями и посмотрите, как будет меняться отображение.



# overflow

Поскольку это достаточно часто является проблемой, стараются не задавать фиксированной высоты блоку (только если это не какой-то графический элемент).



# Сокращённая запись

Достаточно часто для многих свойств используется сокращённая запись:

`padding: 10px` равносильно:

- `padding-top: 10px;`
- `padding-right: 10px;`
- `padding-bottom: 10px;`
- `padding-left: 10px;`



# Сокращённая запись

Узнать, какие конкретно свойства поддерживают сокращённую запись и в каком виде вы можете на странице документации (или сайтах MDN, webref):

## Синтаксис

```
padding: [<размер> | <проценты>] {1, 4}
```

## Значения

Разрешается использовать одно, два, три или четыре значения, разделяя их между собой пробелом. Эффект зависит от количества значений и приведен в табл. 1.

Табл. 1. Зависимость от числа значений

Число значений	Результат
1	Поля будут установлены одновременно с каждого края элемента.
2	Первое значение устанавливает поля от верхнего и нижнего краёв, второе — от левого и правого.
3	Первое значение задаёт поле от верхнего края, второе — одновременно от левого и правого краёв, а третье — от нижнего края.
4	Поочерёдно устанавливается поля от верхнего, правого, нижнего и левого краёв.

Величину полей можно указывать в пикселях (px), процентах (%) или других допустимых для CSS единицах. При указании поля в процентах, значение считается от ширины родителя элемента.

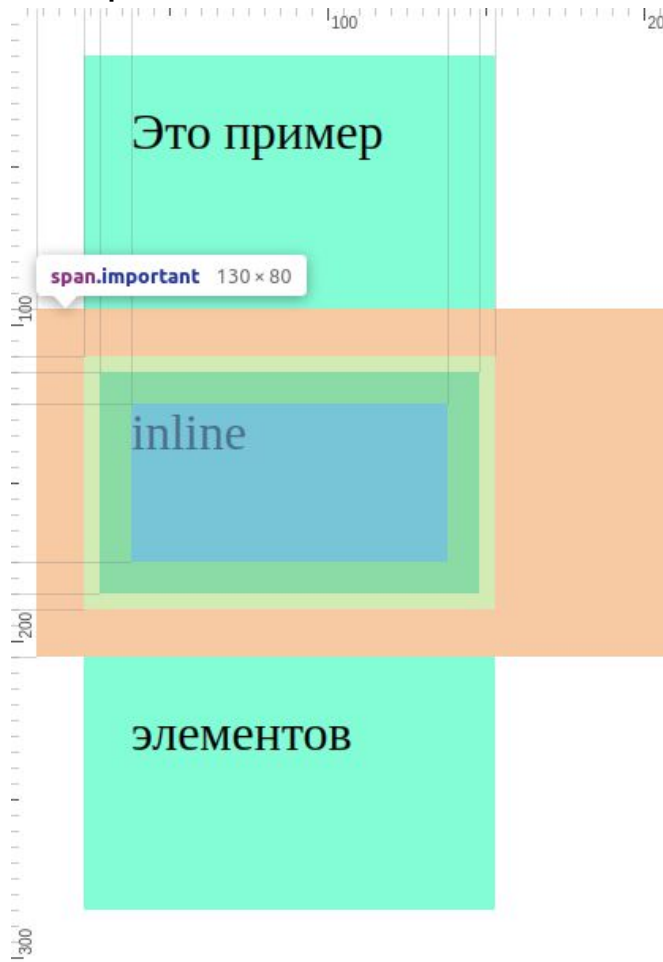




# width & height

`width` и `height` по умолчанию определяют именно ширину содержимого, не учитывая всё остальное. Давайте разберёмся с этим:

```
<style>
  span {
    display: block;
    width: 100px;
    height: 50px;
    padding: 10px;
    border-width: 5px;
    border-style: solid;
    border-color: transparent;
    margin: 15px;
    background-color: aquamarine;
  }
</style>
```

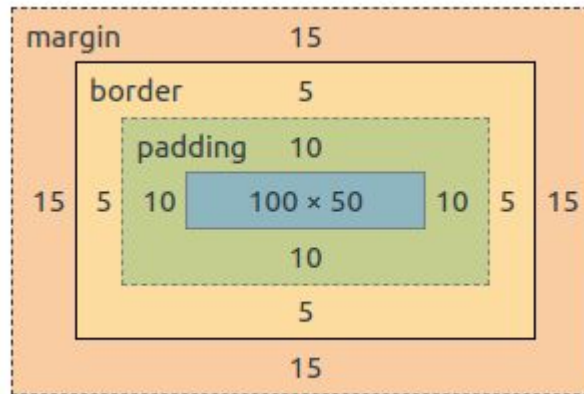


# box-sizing

"Размер" элемента был рассчитан как **130x80**. Т.е. в размеры элемента входят:

- **content**
- **padding**
- **border**

Но не входит **margin**.



Кстати, именно эту область (**content + padding + border**) и заливает **background-color**.



# box-sizing

В большинстве случаев такой механизм не удобен – т.к. нам приходится в голове вычислять сколько будет итоговый размер, складывая все элементы.

За алгоритм расчёта отвечает специальное свойство **box-sizing**:

<i>Name:</i>	<b>'box-sizing'</b>
<i>Value:</i>	content-box   border-box
<i>Initial:</i>	content-box
<i>Applies to:</i>	all elements that accept width or height
<i>Inherited:</i>	no
<i>Percentages:</i>	N/A
<i>Media:</i>	visual
<i>Computed value:</i>	specified value
<i>Canonical order:</i>	per grammar
<i>Animation type:</i>	discrete



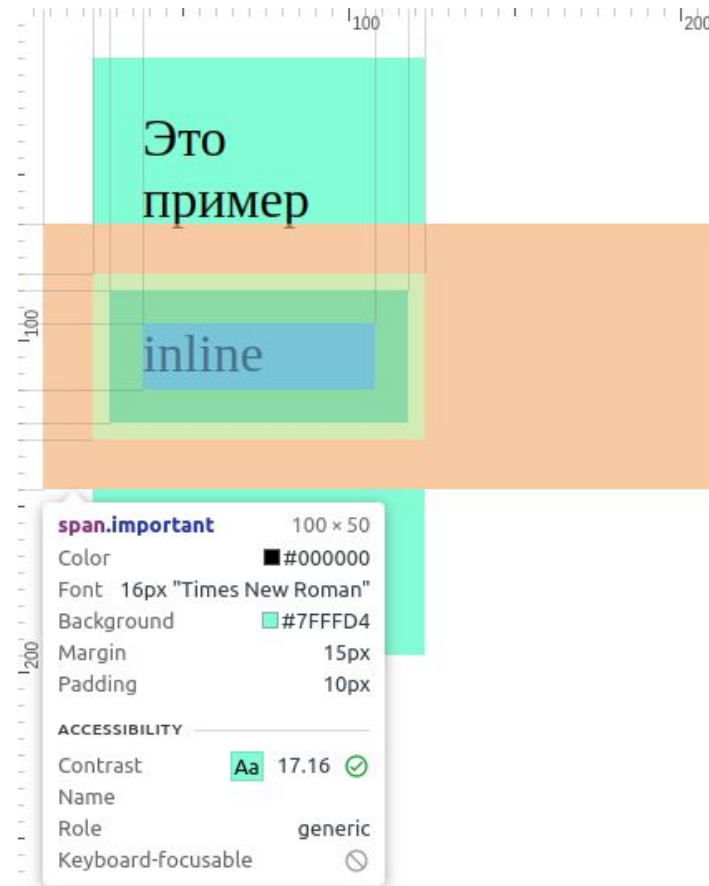
# box-sizing

Попробуем его установить в `border-box`:

```
span {  
  display: block;  
  box-sizing: border-box;  
  width: 100px;  
  height: 50px;  
  padding: 10px;  
  border: 5px solid transparent;  
  margin: 15px;  
  background-color: #7FFFD4;  
}
```

Теперь размеры нормализовались:

- `width = 100px` (и сразу включает в себя и `padding` и `border`, т.е. суммировать ничего не нужно)
- `height` то же самое.





Чаще всего такое правило задают по умолчанию для всех элементов с помощью следующего правила:

```
*, *::after, *::before {  
  box-sizing: border-box;  
}
```



# margin

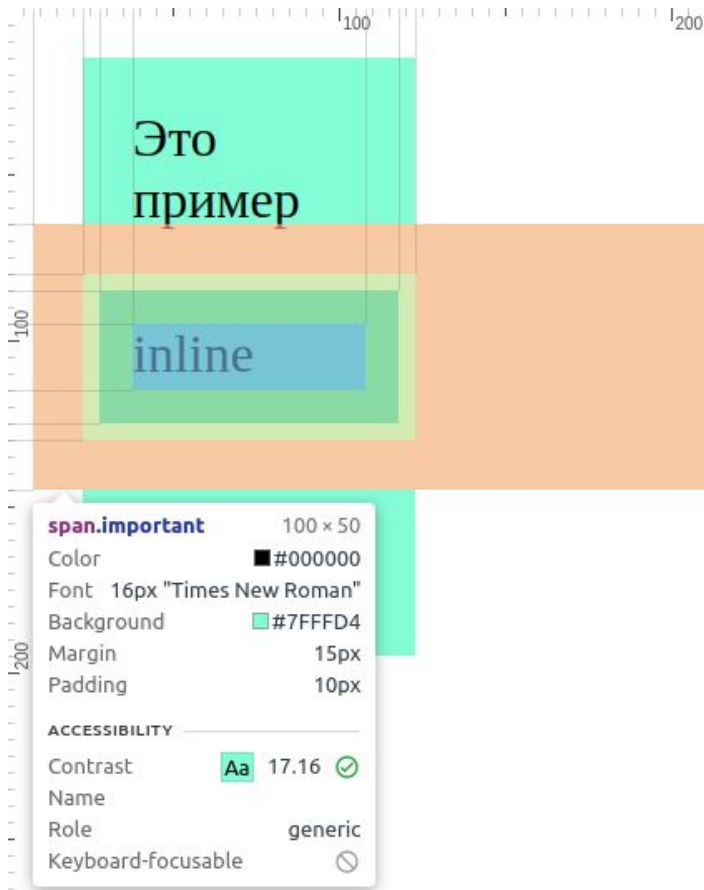
Последнее, что нам осталось обсудить – это **margin** (внешние отступы). Про них важно запомнить две вещи:

1. Они не входят в размеры самого элемента
2. Они "схлопываются" до максимального у двух соседних элементов

Если с первым всё более-менее понятно, то со вторым – не очень. Давайте разбираться.



# margin



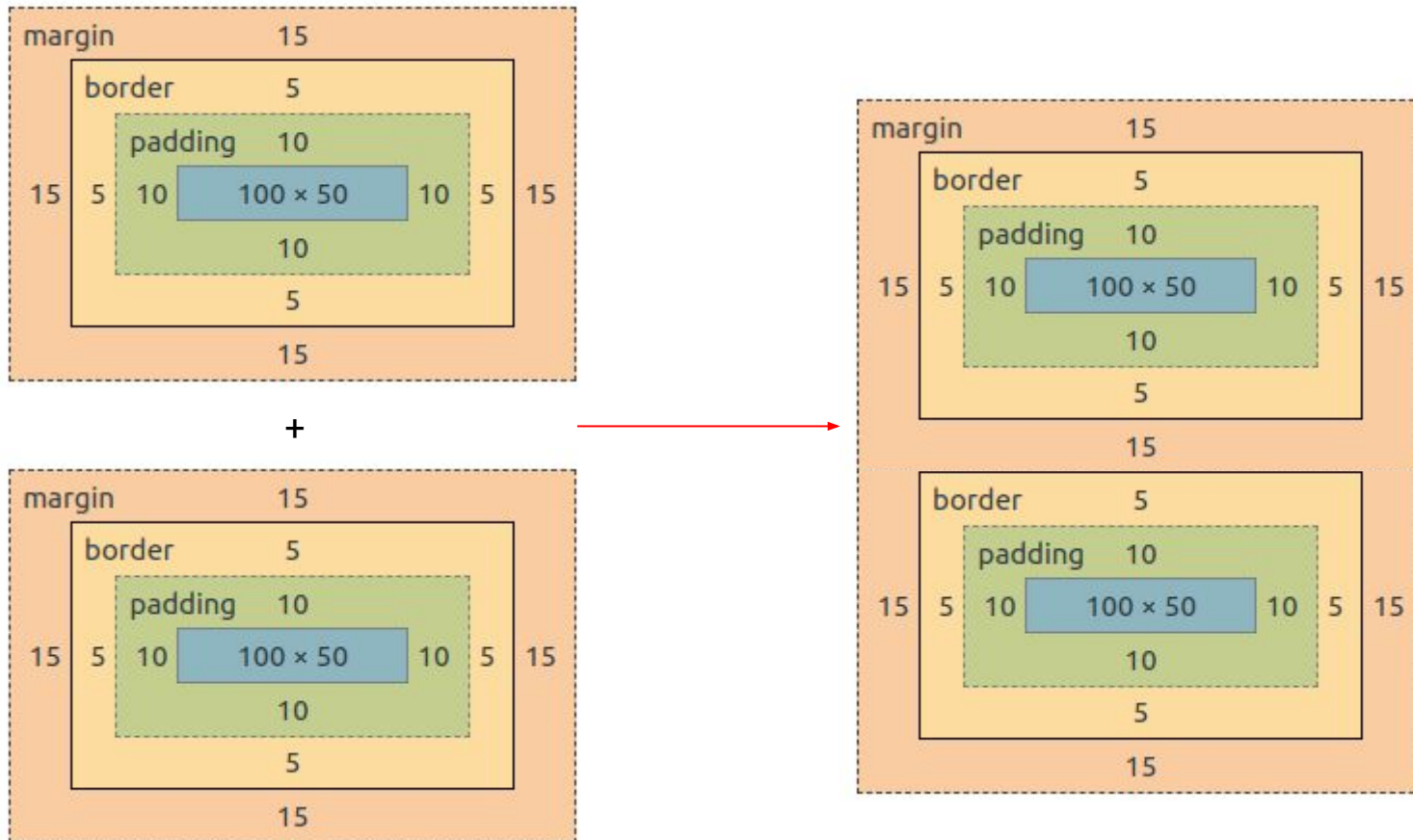
У обоих `span`'ов (у них `display: block`) `margin` = 15px. Поскольку `margin: 15px` – это сокращённая запись, то на самом деле (это видно на вкладке `Styles`):

```
margin: ▼ 15px; ← как записано в CSS
margin-top: 15px;
margin-right: 15px;
margin-bottom: 15px;
margin-left: 15px; | полная запись
```

Но по картинке – расстояние между элементами всего 15px, почему?



# margin



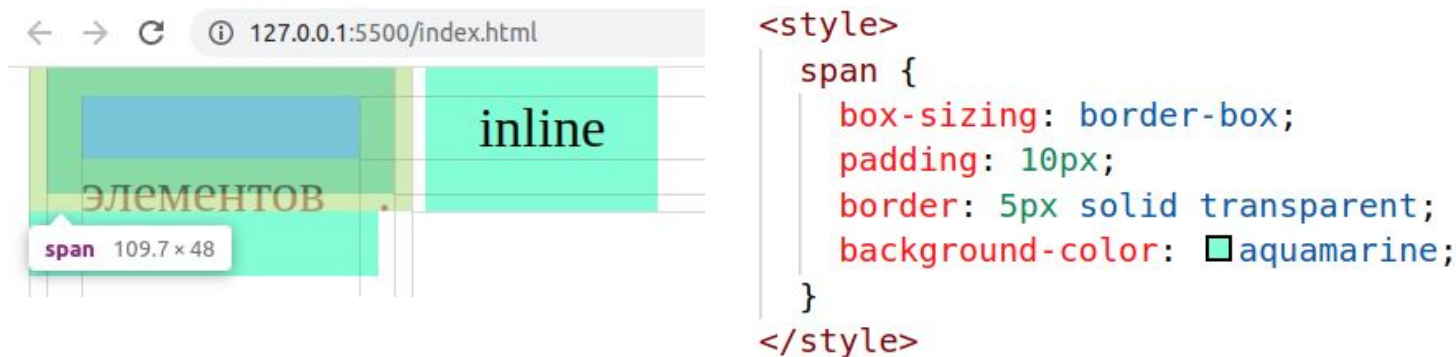
Можете представлять себе это как "элементы не подпускают к себе никого ближе 15px", поэтому между ними устанавливается 15px. Если бы один разрешал подпускать к себе на 15px, а другой – на 20px, то было бы 20px.





# inline

А как же **inline**-элементы? Для **inline** доступна установка **padding** и **border**, но с одной особенностью: установка горизонтальных составляющих будет увеличивать место, занимаемое элементом, а вертикальных – нет:



Как вы видите, по горизонтали – всё хорошо, а вот по вертикали нижний элемент буквально "наезжает" своими **padding**'ами и **border**'ом на вышестоящий.

Поэтому будьте очень аккуратны с установкой вертикальных составляющих.

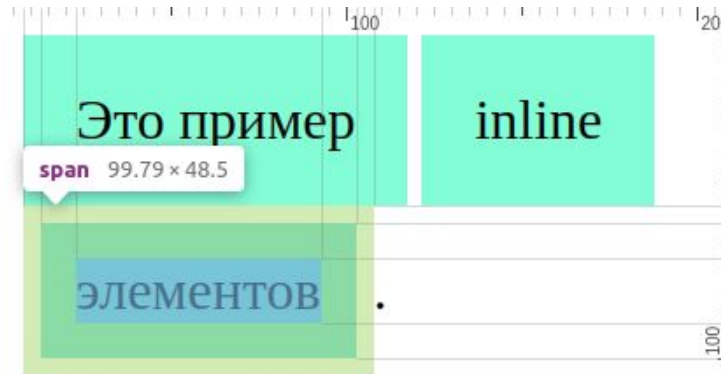


# inline-block

Помимо `inline` и `block` элементов существуют и `inline-block` элементы, сочетающие в себе свойства и тех и других:

- располагаются в строку
- но могут иметь "нормальные" `padding`'и, `border`'ы и даже `margin`'ы

```
span {  
  display: inline-block;  
  box-sizing: border-box;  
  padding: 10px;  
  border: 5px solid transparent;  
  background-color: aquamarine;  
}
```



# Позиционирование



# Позиционирование

По умолчанию, каждый элемент располагается в потоке на своём месте. И все остальные элементы знают, где он расположен и сколько места занимает.

За это отвечает свойство `position`, которое по умолчанию равно значению `static`.

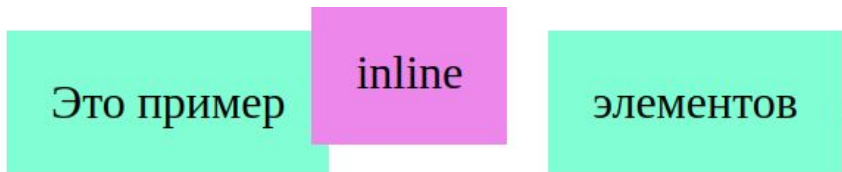
Кроме `static` доступно ещё три ключевых значения:

- `relative`
- `absolute`
- `fixed`



# relative

**relative** позволяет сместить элемент относительно его нормальной позиции в потоке, но при этом все остальные элементы будут "думать", что он по-прежнему занимает своё место в потоке:



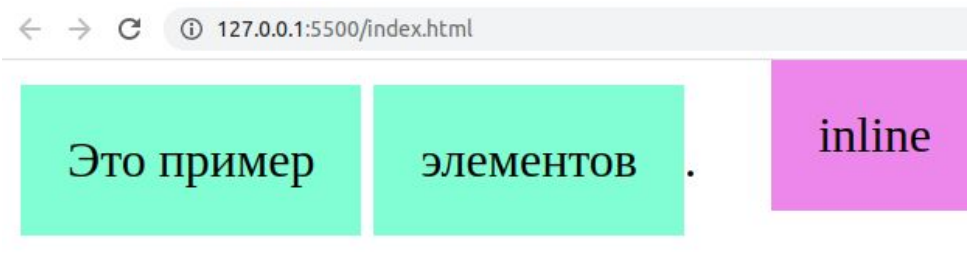
```
.important {  
  position: relative;  
  top: -10px;  
  left: -10px;  
  background-color: purple;  
}
```

Таким образом можно делать всякие смещения, не разрушая поток.



# absolute

**absolute** позволяет "извлечь" элемент из потока (т.е. остальные элементы в потоке будут считать, что этот элемент уже не занимает места) и отпозиционировать его относительно ближайшего родителя, у которого **position** не **static** (либо документа, если таких родителей нет):

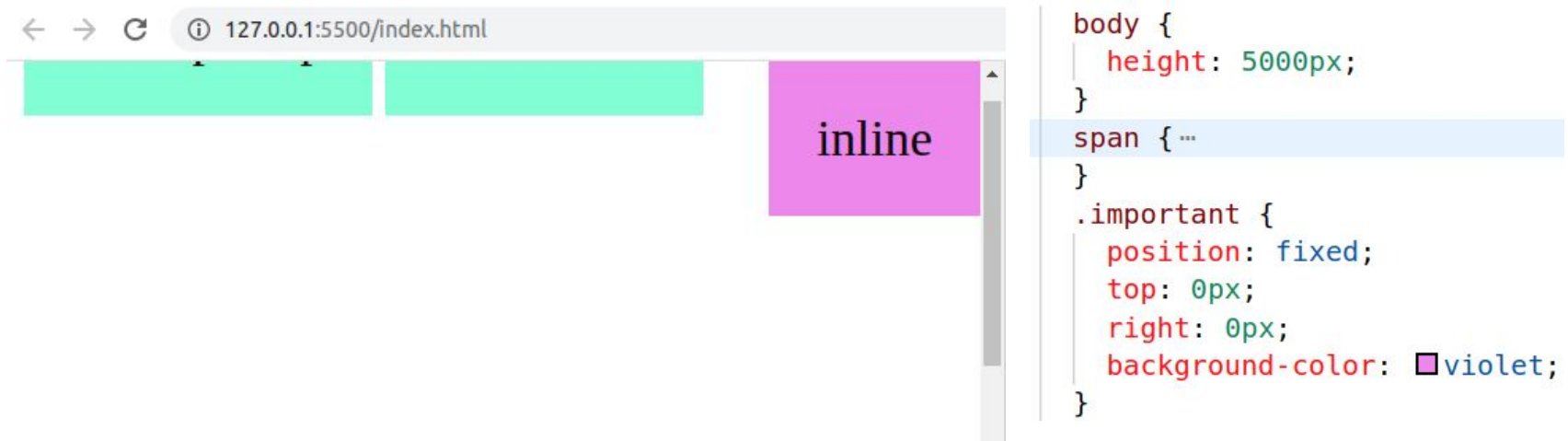


```
.important {  
  position: absolute;  
  top: 0px;  
  right: 0px;  
  background-color: #violet;  
}
```



# fixed

`fixed` аналогичен `absolute`, за исключением того, что позиция фиксируется при прокрутке:



# Итоги





# Итоги

В этой лекции мы глубже познакомились с CSS. После неё у вас должно сложиться впечатление, что не так уж всё и просто с этим CSS (и это действительно так). Но, если мы будем постоянно практиковаться, то быстро запомним правила и сможем двигаться дальше.



# ДОМАШНЕЕ ЗАДАНИЕ



# Орг.моменты

Практикум состоит из 8 обязательных занятий. Начиная с 23 декабря мы выкладываем новые занятия каждый Пн в 10:00 (по Душанбе).

Каждое воскресенье в 23:59 (по Душанбе) дедлайн сдачи домашнего задания.

Если не успеете сдать в срок домашнее задания, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

Все вопросы вы сможете задавать в [Телеграм канале](#).



# Важно

Создайте файл `style.css` (в каталоге `docs`), в котором напишите стили, требуемые в ДЗ.

**Важно:** бот будет искать только `style.css`.



# ДЗ 1: Каскад

У бота будет следующий документ:

```
<body>  
<button class="primary">Записаться на курс</button>  
</body>
```

Создайте файл `style.css` (в каталоге `docs`), в котором напишите стиль, позволяющий переопределить цвет текста кнопки на `#00FF00` с использованием только селектора по классу.

**Важно:** бот будет искать только `style.css`.



## ДЗ 2: Цвет кнопок

У бота будет следующий документ:

```
<body>
  <header class="header">
    <a href="https://alif-skills.pro/login" class="button">Войти</a>
  </header>
  <footer class="footer">
    <a href="https://alif-skills.pro/cooperate" class="button">Сотрудничать</a>
  </footer>
</body>
```

Используя только CSS-селекторы по классам (и их комбинации), задайте кнопке **Войти** фоновый цвет прозрачным. Предполагается, что где-то выше стиль кнопки объявлен как (вам это писать не нужно):

```
.button {
  background-color: #00BFBF;
  color: #FFFFFFF;
}
```



# ДЗ 3: Звёздочки

У бота будет следующий документ:

```
<body>
  <p>В программе:</p>
  <ul>
    <li class="featured">HTML</li>
    <li class="featured">CSS</li>
    <li class="featured">JS</li>
  </ul>
</body>
```

Используя только CSS-селекторы по классам, сделайте так, чтобы вывод был таким (в качестве символа используйте: ★):

В программе:

- ★HTML
- ★CSS
- ★JS



## ДЗ 4: hover

У бота будет следующий документ:

```
<body>  
<button class="primary">Записаться на курс</button>  
</body>
```

Используя селектор по тегу и псевдоклассы, напишите правило, по которому при наведении указателя мыши на кнопку фоновый цвет кнопки станет #0027A0.





# ДЗ 5: position

У бота будет следующий документ:

```
<div class="image">  
    
  <button class="like">♥</button>  
</div>
```

Используя селекторы:

- для `div` – только по классу
- для `button` – комбинацию по классу элемента и по классу его родителя



# ДЗ 5: position

Важно:

1. Сначала пишете правило для родителя (`div`)
2. Затем для `like` (кнопки)



Спасибо за внимание

**alif skills**

2022г.

